Express Mail Label No. EV 318163395 US

[0001] SOFTWARE PORTING LAYER

[0002] CROSS REFERENCE TO RELATED APPLICATION(S)

[0003] This application claims priority from U.S. provisional application no. 60/405,995, filed August 26, 2002, which is incorporated by reference as if fully set forth.

[0004] FIELD OF INVENTION

[0005] This invention generally relates to wireless devices. In particular, the invention relates to developing software to run on hardware of such devices.

[0006] BACKGROUND

[0007] Developing software to effectively use hardware, such as communication processors and system processors, of wireless devices is difficult. Wireless devices have limited memory storage capacity, making it desirable that software operating systems, interfaces and applications have a small footprint. However, using reduced footprint software design techniques limits the flexibility accorded to software designers.

[0008] To illustrate, one tool for producing software modeling language SDL code is Telelogic Tau. That tool allows for software programmers to produce a model of the software and the tool produces code from the model. Telelogic Tau has two potential code generators, C-advanced and C-micro. C-advanced allows a software designer to use all the features of SDL. C-micro limits the available features with the advantage of producing a smaller code footprint.

[0009] Telelogic Tau supports both light and tight integration. In light integration, the entire SDL system executes in a single thread. In tight integration, each SDL process becomes a thread. To minimize the code footprint, C-micro is desirable to be used. However, C-micro only supports light integration and, accordingly, multiple threads of execution are not supported. Accordingly, it is desirable to have code with a reduced footprint that supports multiple threads of execution.

[0010] SUMMARY

[0011] A porting layer takes software developed using a single threaded modeling tool to a multiple threaded environment. The single threaded modeling tool is used to model the software. The porting layer ports in variables into a multiple threaded operating environment by reference and not as variables so that each thread can access variables by reference.

[0012] BRIEF DESCRIPTION OF THE DRAWING(S)

[0013] Figure 1 is a simplified diagram of one configuration of a wireless device with software and programmers interface illustrated.

[0014] Figure 2 is an illustration of messaging between SDL processes using light integration across partitions.

[0015] Figure 3 is an illustration of messaging between SDL processes within a partition using tight integration.

[0016] Figure 4 is an illustration of messaging between SDL processes across partitions using tight integration.

[0017] Figure 5 is an illustration of the structure of SDL C-micro generated data types.

[0018] Figure 6 is a list of preferred reserved SDL process identities.

[0019] Figure 7 illustrates the format of an SDL process control block.

[0020] Figures 8, 9 and 10 are flow charts of a signal identity translated into a transition identity within the SDL porting layer.

[0021] DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT(S)

[0022] Figure 1 is a simplified diagram of one configuration of a wireless device with software and programmers interface also illustrated. One such wireless device is a wireless transmit/receive unit. A wireless transmit/receive unit (WTRU) includes but is not limited to a user equipment, mobile station, fixed or mobile subscriber unit, pager, or any other type of device capable of operating in a wireless environment. Although the preferred embodiment uses the hardware configuration of Figure 1, alternate hardware configurations can be used with the wireless device software.

[0023] The wireless device has a system processor 100, such as a RISC. The system processor 100 typically runs the system applications, such as web browsing, calendars, etc. A communication processor 102, such as a DSP, typically runs the wireless communication protocols. A communication module 104 allows for communication between the system and communication processor. A shared memory 106 is accessible by both processors 100 and 102 via the communication module 104.

[0024] Figure 1 also illustrates the preferred software structure of the wireless device and its association with the hardware. The SDL model layer 108 represents the communication protocol stack components at very high level of abstraction. The SDL model environment 108 is composed of SDL processes 118_1 to 118_6 (118) communicating at a peer-to-peer level via SDL signals. SDL model elements (processes and signals) are transformed into operating environment elements by targeting the code generator to the operating system adaptive port interface (OS API) 120.

[0025] The SDL porting layer 110 represents the conversion of SDL model elements (processes, signals, partitions, etc.) into operating environment constructs through the OS API 120. The operating environment 112 provides an OS/hardware independent environment via the OS API 120. The OS abstraction layer 114 implements the OS API 120 for a particular underlying operating system and hardware platform. It maps the OS API 120 constructs into their underlying OS representations.

[0026] The OS abstraction layer 122 interfaces with the communication module 104 to support inter-processor communication. The communication module 104 provides the communication path between the OS API threads 128_1 to 128_2 (128) across the system/communication processor boundary and is responsible for the transfer for both data and control. As shown in Figure 1, a queue 130_1 to 130_2 (130) is associated with each thread 128. The communication module 104 provides the communication path between OS API threads across the system/communication processor boundary. It is responsible for the transfer of both data and control. A system OS 124 is associated with the system processor 100 and the communication OS 126 is associated with the communication processor 102.

[0027] Although, all of the layers work together in the preferred software structure, each layer can be typically used with other layers in alternate software structures.

[0028] Figures 2 to 4 illustrate of communication between SDL processes 118. Figure 2 is an illustration of messaging between SDL processes 118 using light integration. SDL processes 118 execute on top of an SDL Kernel (K/E) 132_1 to 132_2 (132). The SDL kernel 132 manages communication between the processes 118, routing signals to their appropriate recipients. The SDL model 108 may be divided into separate light integration partitions. Each partition contains its own SDL kernel 132, and resolves to a single thread 128 with a queue 120 within the operating environment 112. Each partition's SDL kernel 132 manages communication between internal SDL processes 118 without OS assistance. The SDL kernel 132 forwards signals destine for SDL processes 118external to its partition through the operating environment 112via messages 134_1 to 134_2 (134).

[0029] Initially, the signal from one SDL process 118_2 is converted to an OS message by the SDL kernel 132_1 in a first environment. The message is routed to an SDL kernel 132_2 in a second environment. In other words, the message is "put" in the second environments queue 130_2. The SDL kernel132_2 in the second environment detects the message, and converts the received message back to its signal format. The second environment SDL kernel 132_2 signals the appropriate process.

[0030] Fundamentally, each SDL light integration partition resolves to a thread with a single message queue within the operating environment. Communication between partitions occurs via messaging. Each SDL kernel / environment knows what SDL processes 118 are internal, and what SDL processes 118 are external. Each SDL kernel / environment knows where to route external signals (i.e. what partition their recipient resides) and each SDL kernel / environment knows how to convert signals to messages, and vice versa.

[0031] Figure 3 is an illustration of messaging between SDL processes 118 within a partition using tight integration. Using tight integration, the SDL model 118 maps directly to the operating environment 112. All SDL process communication (signaling) takes place through the operating environment 112. The SDL kernel (SDL process scheduler) is not included using tight integration. Specifically, each SDL process resolves to a thread 128 with a single message queue 130 within the operating environment 112. Communication between SDL processes 118 takes place through the operating environment 112. The SDL porting layer 110 converts signals to messages 134, and vice versa.

[0032] For a first SDL process 118_1 to signal a second SDL process 118_2, the first process 118_1 converts the signal into an OS message. That message is routed to the second SDL process thread 128_2 via its message queue 130_2. That SDL Process thread 128_2 wakes up on the message and the message is converted back to a SDL signal. The second SDL Process 118_2 receives the signal from first SDL Process 118_1.

[0033] Figure 4 is an illustration of messaging between SDL processes across partitions using tight integration. For a first SDL process 118_1 to signal a second SDL process 118_2, the first process 118_1 converts the signal into an OS message. That message is routed to the first partition external task 136_1. This environment is the environment acting as a proxy for all processes external to this partition. That partition external task 136_1 forwards the message (signal) to the second partition External Task 136_2. The second partition external task 136_2 routes the message (signal) to the second process thread 128_2 via its message queue 130_2. That thread wakes up on the message. The message is converted back to a SDL signal and the second SDL process 118_2 receives the signal from the first SDL process 118_1.

[0034] This procedure facilitates talking across processor boundaries. The partitions can by targeted to a specific processor and to function across the boundaries external tasks 136 may be used.

[0035] Abbreviations

ADT - Abstract Data Type

API - Application Programmer's Interface (Client Interface in this context)

CPU – Hardware (HW) Processor: system RISC (such as an ARM) or DSP.

IPC - Inter-processor Communication

OS - Operating System

HAL - Hardware Abstraction Layer

HW - Hardware

MUX - Mutex

SoC - System-On-a-Chip

SW - Software

[0036] Terms and Definitions

Communications Processor - Embedded CPU (DSP) executing low-level communication protocol stack layers, algorithms, device drivers, etc.

Operating Environment - Abstract programming environment provided by the OS abstraction layer through the OS API.

Open Platform - Operating system with a published (or publicly available) API. Third party organizations develop user-level applications to the published API.

Proxy - Placeholder or surrogate for another object, controlling access to it. Used in the OS Abstraction to present constructs physically located on a remote processor.

System Processor - General-purpose CPU executing an "Open Platform" operating system or a real time operating system (RTOS) including user-level applications.

Underlying Operating System - A target OS on which the OS abstraction layer and the OS API are built.

[0037] In the preferred embodiment, an SDL porting layer 110 is used. The preferred SDL porting layer allows for software modeled using a single threaded code generation tool to run as multi-threaded software. The most preferred application is to code generated using C-Micro light integration of Telelogic Tau. The SDL porting layer 110 effectively converts the light integration (single thread) to a tight integration (multiple threads). The most preferred application allows for reduced code footprint (C-micro, light integration) with the added benefits a multi-threaded environment.

[0038] In a typical single threaded operating environment, the variables are brought in as global variables. Since the variables operate in a single threat, there is only a need for global variables. To facilitate operating in a multi-threaded environment using single threaded modeling tools, the variables are brought in as parameters/by reference and not global variables. The transition functions are modified to allow this functionality. This functionality prevents one thread from stopping on another threads variable. This facilitates the use of parameter variables by independent threads. The preferred structure to facilitate this is described herein.

[0039] In other embodiments, the other tools can be used without the SDL porting layer. Additionally, aspects of the SDL porting layer can be used with other single thread code generation tools as well as multi-thread code generation tools other than C-micro.

[0040] The basic requirement of the porting layer 110 is to produce the same SDL system behavior as the SDT provided C-micro kernel. This includes the ability to support the following SDL model features (SDT-imposed restrictions in parentheses).

SDL Feature:

- Multiple processes
    - Send-Via signals (no output via all)
    - Send-To signals
    - Deferred signals
    - Multiple instances of a process (finite number only)
    - Asterisk input, asterisk state
    - Timers (integer duration only)
    - Signal w/ parameters (no omission of parameters in a signal input)
    - Timer w/ parameter (one integer parameter only)
    - Dynamic process creation (no formal parameters)
    - Procedure (no inheritance, no states, no nested procedure call data scope)
    - Service
    - Predefined Types
    - (No RPC)
    - (No Service and Priority Input and Output)
    - (No Export/Import, View/Reveal)
    - (No Enabling condition / Continuous signal)
    - (No Any Expression)
    - (No Macros)
    - (No channel substructure)
    - (Names of processes within different blocks must be different)

[0041] A number of options may be available during SDL code generation. The following description is based on the Make window in SDT to perform this step, although others may be used.

[0042] The options chosen for the preferred code generation are as follows, although others may be used.

Analyze & generate code (only)

- Code generator: C-micro
- Prefix: Full
- Separation: Full
- Capitalization: As defined

[0043] The code generated by SDT consists mainly of data structures. The only pieces that translate to operational code are the transition functions associated with each process. The structure of the generated data types is represented in Figure 5.

[0044] The root process table 138 is a global array, sized to the maximum number of process types in the system, plus one to allow for an end marker. This array is composed of process description block entries, one for each process type in the SDL system. The contents of each process description 140 are static variables, placed in the process's source file, along with the transition function. The root process table 138 is placed in the system source file.

[0045] The enumerations (process type and timer / signal IDs) are placed in header files 142. Process types are preferably numbered beginning at 0. To accommodate multiple instances of a single process type running in the system, an instance number is combined with the process type to come up with the process ID, or PID. Instance numbers begin at 0. Timer ID's and signal ID's share a common enumeration. Timers ID's are defined first, beginning at 1 (0 is reserved for the Start signal). Figure 6 is a list of preferred reserved IDs.

[0046] An SDL system can be partitioned using either automatic or manual partitioning. Automatic partitioning is accomplished preferably via an SDT build script. When the system is partitioned, send-via signals that cross a partition's boundary get translated into signals that are sent-to an "environment" SDL process. The identity of the true intended recipient is lost. Enough information may be available in the SDT-generated PR file to construct a mapping table (signal ID + sending process ID = destination process ID), which the environment process could use to forward the signals. However, this imposes more restrictions on the design of the system and adds extra hops to the signal's route.

[0047] Manual partitioning is done after code generation. A copy is made of the system source file, and the two are edited. The root process table entries for processes not in the partition are replaced with null entries. Next, the files are grouped according to partition and compiled using an appropriate make file. Full separation is required during the code generation step so that this grouping can occur. The main drawback to manual partitioning is the need to edit a generated code file. However, it produces more efficient code, and it does not impose limitations on the system.

[0048] For SDL system startup, each partition within the SDL system is compiled into a single executable (exact form is both OS and HW platform dependent). At startup, the partition's "main" (main thread) loops through the root process table 138. Each non-null entry represents a local process type. The process description 140 for each process type contains an instance data table, sized to the maximum number of instances of the type that can be running simultaneously in the system. A thread group is created, containing a synchronized thread for each possible process instance. A process control block is created for each instance, containing the process ID and a pointer to the process description. A pointer to the process control block is passed to the thread so that it has scoped access to its own information.

[0049] The thread group is directed to synchronize, then to start. During synchronization, each thread creates for itself a message queue, a save list, and an active timer list. The handles for these items are stored in the process control block. If a process instance should be active on startup, as dictated by the generated code, a thread then sends a Start signal to itself. At this point, initialization is complete, and the thread notifies the thread group accordingly. When all threads have checked in, the thread group can be started, and active instances will receive and process their Start signals.

[0050] Figure 7 illustrates the format of a Process Control Block. The data type and its associated table are extensions to C-Micro to support tight integration to the OS API. The process control block 144 has a PID 154, OS handles 156_1 to 156_3 and a process description pointer 158. One OS handle 156_1 points to a message queue. Another OS handle points to a save list 148 and another 156_3 to an active timer list 150. The process description pointer 158 points to a process description 152.

[0051] For SDL signal inputs, each SDL process instance runs in its own operating environment thread. Its responsibility is to receive and react to signals. Signals are received in messages through the thread's associated message queue. There are three ways an SDL process may react to a signal: discard it, save it for later, or initiate a state transition.

[0052] Once a signal has been identified as the current signal, the SDL process decides how to react to it. The framework for processing a signal resides in the porting layer and uses data accessed through the process control block 144. The following is the processing of the signal.

[0053] As illustrated in the flow chart of Figures 8, 9 and 10, the signal ID is translated into a transition ID. The process ID 154, stored in the process control block 144, provides the instance number. The instance number is used to index into the state index table 160 in the process description 152, and finds the current state. If the current state is Dormant 162, the transition ID is set to discard 164. If the current state is Start and the signal ID is Start, 166, 170, the transition ID is set to Start and the process returns to Start, 176. Any other signal received while in the Start state results in a transition ID of Discard, 174.

[0054] If the current state is neither Dormant nor Start, the signal is checked to see if it represents a timer, 178. When a timer is set, the sender ID of the signal is set to a reserved Timer process value, and an entry is added to the active timer list 180. If a timer is cancelled, its entry is removed from the list. If a signal 172 is received whose sender has been set to the reserved Timer process value, it is identified as a timer signal. If an entry for that timer is found on the active list 180, 182, the entry is removed and processing continues. If it is not found, the timer is invalid, and the transition ID is set to Discard, 184.

[0055] If the transition ID has not yet been set, the expected signals 188 for the current state are checked. In the state index table, each state maps to an index into the transition table, 190, where the signals expected in that state begin. The transition table entries are state-ordered the same as the state index table, so by also using the subsequent state's index, a range of indices into the Transition Table 190 is obtained. Within this range, if a signal ID is found that matches the current signal 188, or if the Asterisk signal ID is defined for this state, the corresponding transition ID is taken, 194.

[0056] If a valid transition ID has still not been found, the Asterisk State is checked. If the Asterisk State is defined for this process, the first entry in the state index table is not a 0, since the Asterisk state entries in the transition table come first. Therefore, this bottom range is checked the same as the current state.

[0057] If the transition ID is Discard, 198, or if all attempts to obtain a valid transition ID fail, a Null, (the signal is unexpected), the signal is discarded, 208. If the transition ID is "Save", 200, the signal is saved via advance itorator 210, (on the save list 212). All other transition ID's are fed into the process description's transition function 214, the behavior of which is defined in SDL, and a new state ID is returned.

[0058] If the new state is not Dash, 202, the instance state table 216 is updated to the new state. If the new state is Dormant, 206, all active timers are cancelled, 218, 220. The save list 212 and the message queue are not flushed, 204, but are handled through normal processing. If the process is quickly started again, this may result in stale signals reaching the new process (since Start signals are sent with High priority and could jump ahead of stale signals in the queue). However, this is preferable to mistakenly discarding signals intended for the new process.

[0059] For signal output, SDL processes send signals from within the transition function. A connection is acquired to the destination message queue. Since opening and closing a connection to a message queue each time a message is put on it is undesirable, connections are opened as needed, and kept in a shared area for all threads in the process to use until system shutdown. This is accomplished via a lookup function, provided by the OS abstraction layer 122.

[0060] Timers can be set by an SDL process. When a timer goes off, it is received and processed as a signal. An SDL process can also cancel timers. Once a timer has been cancelled, it should not appear on the incoming message queue.

* * *